A Discussion of Optimization Strategies and Performance for Unstructured Computations in Parallel HPC Platforms

D. Shires and R. Mohan*and A. Mark
High Performance Computing Division
U. S. Army Research Laboratory
Aberdeen Proving Ground, MD

Abstract

This paper describes the optimization strategies employed and their degree of success for a parallel finite element application for liquid composite manufacturing process modeling. This code represents a difficult challenge for both the software developer and the parallel computer alike. The unstructured nature of the code creates an environment requiring both large amounts of computation and communication. This in turn provides opportunities to judge the true performance of large-scale parallel systems and also the effectiveness of various parallel development methodologies. We discuss the use of message passing (using the Message Passing Interface) in this code and various optimization strategies used to increase the code's performance. We also compare this performance against those results achieved from a data parallel implementation using High Performance Fortran (HPF).

1 Introduction and Motivation

The development of the multiprocessor has opened up new possibilities in the realm of scientific computing. However, the variations and complexities of the parallel computers built today require careful code profiling and tuning to achieve the maximum performance possible. Achieving good performance also requires a comprehensive understanding of the various parallel methodologies available.

One such methodology is message passing where processors communicate through explicit calls to communication routines. The Message Passing Interface (MPI) has become the *de facto* standard for achieving effective scalable parallel software implementations for this

*Dr. Mohan, an employee of the University of Minnesota, is currently stationed at the U. S. Army Research Laboratory.

paradigm. Scalable codes written using MPI are extremely portable and are applicable to clusters, shared memory architectures, and massively parallel computing platforms.

Another approach to parallel computing is data parallelism. Data parallelism implies that simultaneous operations are applied to every item in a large data set. Implemented correctly, this approach is also germane to the various parallel computing architectures in use today.

This paper details our experiences in developing parallel computing approaches for the COMPOSE (Composite Manufacturing Process Simulation Environment) suite of codes currently under development at the U. S. Army Research Laboratory and the University of Minnesota. These codes are part of the Department of Defense High Performance Computing Modernization Program, Common High Performance Computing Software Support Initiative (CHSSI). We also briefly address the use of HPF to achieve a parallel solution. These codes are designed to enable large-scale liquid composite manufacturing simulations. By combining parallel computing and novel algorithmic approaches, previously impossible large-scale simulations and even parametric studies are now possible. These simulations, that have wide applicability to Army future combat systems and commercial ventures, provide a simulation based approach for reducing cost, time, and risk associated with the composite manufacturing industry.

The ability to provide timely process models and simulations is critical for optimization of an integrated and intelligent manufacturing environment. Accordingly, we have vested significant effort in selecting the most efficient strategies possible when scaling to large computing architectures. Furthermore, significant effort has been spent to optimize the software to provide the fastest solution possible. The following sections further describe the problem being solved and the message passing and data parallel solutions to the problem. We concentrate on the optimizations and their effects in the MPI version of the code. We conclude by comparing the performance of the two approaches with some general observations. It should be noted that these developments are applicable to the general set of unstructured finite element codes.

2 Computational Problem

The primary objectives of the COMPOSE suite are to predict resin impregnation behavior in fiber preforms and to track the pressure field and flow front during filling. Other factors, such as temperature effects, are in various stages of development and testing. Liquid composite molding processes require a permeable fiber preform to be saturated with a thermosetting resin. Failure of the resin to completely imbue the preform will result in a failed manufacturing attempt. Eulerian fixed-mesh approaches are employed to model the various complexities of the parts and molds. Finite element meshes comprised of nodes and elements (nodal connectivity) are used for the computational model. Unstructured meshes, where any number of elements can meet at a single node, are required for these geometrically complex models. More detailed descriptions of the computational problem and pertinent applications are available [1–3].

3 MPI

3.1 Approach

Current core solver developments use the Fortran 90 programming language. This language was chosen due to its many enhancements over the FORTRAN 77 language. Predominately, the new feature of most value is the ability to use heap storage through allocatable arrays. Because of the uncertain impact on performance of more advanced features, such as abstract data typing, these are used more rarely and not in areas of compute bound loops. Pre- and post-processing routines largely employ software written in C++. Solver translations into C++ are planned for the future to allow the code to be more tightly coupled with pre- and post-processing developments and user interfaces. Furthermore, tremendous advantages in software reuse have been noted in the various preprocessing routines, and we seek to carry this over to the numerical solver as well.

A domain decomposition approach is required to make the problem suitable for SPMD parallelism. This step produces a partition of the larger problem into a set of processor-specific, nonoverlapping smaller subdomains. Figure 1 shows a finite element mesh partitioned for 8 processors. Each of these subdomains is in turn mapped to a processor to allow for a parallel solution. Shared node boundaries present a problem in that they represent areas where explicit communication between the processors is required. A good mesh partitioner is obviously one that tries to promote load balance by creating subdomains that are roughly the same size, and also one that tries to reduce the number of nodes that are shared between the subdomains. Several tools are available to create these partitions, including METIS (on which our partitions are based) and JOSTLE [4, 5].

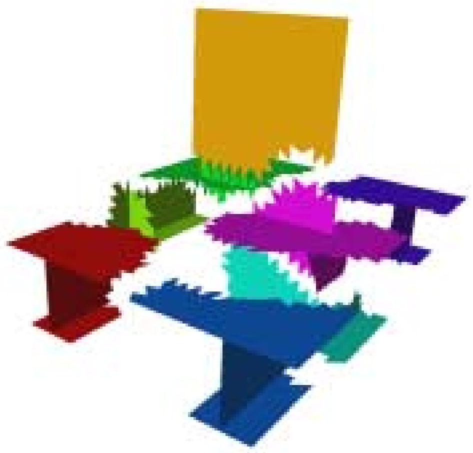


Figure 1: 8-processor domain decomposition of an airframe structure.

Several guiding principles are in order for explicitly parallel codes using message passing. To begin with, the removal of any unnecessary barriers is paramount. Detailed control flow analysis early in code development revealed an unnecessary call to `MPI_BARRIER`. Analysis indicated that program flow would result in all processors executing a reduction before any critical region, hence the barrier could be removed. Furthermore, we attempted to hide communication behind computations. This is well known in library-based optimizations where the compiler is not free to optimize through the call. However, we were limited in this regard because of the tight loop structure of the solver and update sections of the code. We enabled the system to be more fluid by using nonblocking sends and wildcards, thus limiting any artificial requirements for data communication ordering.

Domain decomposition leads to numerous interface nodes that are shared between various subdomains. Also, numerous matrices and vectors are formed that are local to the various processors. These can be in either local or assembled forms. We employ a preconditioned conjugate gradient solver to compute the vector-vector inner products during the iterations. Such an approach is efficient, but heavily taxes the processor interconnection network. Interface vector entries are sent and received using the send and receive pairs mentioned previously. The `MPI_ALLREDUCE` call is used to determine the global inner product from the subdomain inner products. It should be kept in mind that this call is not only nonscalable, but actually creates more of a burden to the system as processors are added.

```
• ${x_0^{SD}} = 0$, ${r_0^{SD}} = {b^{SD}}$
• Do $i = 0, 1, \ldots\ldots$ until convergence
    1. If $i = 0$ Go To Step 5
    2. $\{u^{SD}\} = [A^{SD}]\{p^{SD}\}$
       $\beta^{SD} = \{p^{SD}\}^T\{u^{SD}\}$
       $\sigma_{SD} = \beta^{SD}/\gamma$
    3. MPI_ALLREDUCE $\frac{1}{\alpha} = \Sigma\sigma^{SD}$
    4. $\{x^{SD}\} = \{x^{SD}\} + \alpha\{p^{SD}\}$
       $\{r^{SD}\} = \{r^{SD}\} - \alpha\{u^{SD}\}$
    5. SEND $\{r^{SB}\}$; RECEIVE $\{r^{NB}\}$
       $\{\hat{r}^{SD}\} = \{r^{SD}\} + \Sigma\{r^{NB}\}$
       $\rho^{SD} = \{\hat{r}^{SD}\}^T\{r^{SD}\}$
    6. MPI_ALLREDUCE $\gamma_{new} = \Sigma\rho^{SD}$
    7. If $(i = 0)$ Go to Step 9
    8. If $\left(\frac{\gamma_{new}}{\gamma} < \epsilon\right)$ STOP
    9. $\{p^{SD}\} = \{\hat{r}^{SD}\} + \frac{\gamma_{new}}{\gamma}\{p^{SD}\}$
    10. $\gamma = \gamma_{new}$
```

Figure 2: MPI Solver for COMPOSE.

The solver in COMPOSE requires the solution of a linear system of equations based on a sparse, symmetric positive definite matrix of the form $Ax = b$. An iterative conjugate gradient algorithm, developed in the framework of multiple instruction, multiple data (MIMD) computers, is used [6]. Diagonal preconditioners were later used to improve the convergence rate [7]. The iterative procedure is shown in figure 2. Entries representing assembled forms require communications during formation. With many scientific codes it is common to observe a common 90/10 ratio for execution; 90% of the computations are executed in about 10% of the code. This code exhibits this behavior and this small code segment represents the majority of the execution time. The computational data structures involved in this solver consist of both node and element level computations, but a coupled interdependency exists for these two levels of computations. Hence, this code can quickly go from being compute bound to communication bound for large sets of processors.

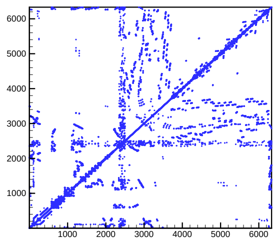
3.2 Optimizations

As previously stated, MPI uses the SPMD approach to parallelism. Accordingly, code tuning for parallel performance is simply an extension of optimizing the single thread execution. We have also noted performance gains by modification of data sets that does not effect the physics behind the simulation.

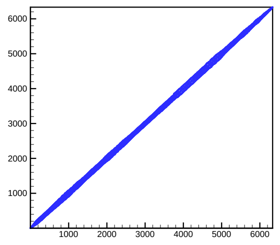
3.2.1 Data Optimizations to Improve Cache Performance

Most reduced instruction set computers (RISC) use various levels of cache to overcome the CPU/memory system performance gap. Unfortunately, the unstructured nature of the meshes used in COMPOSE can cause problems with maintaining cache affinity. That is, data will often be loaded into cache only to be swapped out quickly. Easily, the largest cost of computing in the COMPOSE solver comes from repeated matrix-vector multiply operations. The node-based computations with the matrix A, stored in compressed row format, are based on element connectivity lists. For example, if a triangular element is comprised of nodes 1, 2, and 3231, a load of data for node 1 will also load data for node 2 into cache, but mostly likely not for node 3231. Ideally, we would like to have this element comprised of nodes 1, 2, and 3.

Various techniques have been developed with the intent of reducing the bandwidth and envelope of a sparse matrix representation and hence improving memory system performance [8]. We have implemented a technique based on the Cuthill-McKee (CM) approach [9]. A slight modification of reversing the ordering (RCM) has been shown to be more efficient and has been added to our preprocessing pass. Figure 3 graphically displays the non-zero entries for an unstructured mesh. Figure 3(a) shows the structure based on the unprocessed mesh coming from a commercial Computer Aided Design (CAD) package. Figure 3(b) shows the same mesh after being reordered. The difference in this case was quite dramatic.



(a) Before reordering



(b) After reordering

Figure 3: Distribution of non-zero entries in a finite element sparse matrix.

How well did bandwidth reduction reduce cache misses and effect performance? We have observed significant gains on the Cray T3E-1200 system. This architecture contains only 8 KB of primary cache and only 96 KB of combined secondary data and instruction cache. The RCM processed meshes consistently ran about 10% faster than the original meshes. However, the large-cache SGI Origin 3800 (8 MB secondary cache) architecture ran each mesh in roughly the same time. A study of the hardware performance counters on this machine did reveal some memory system improvements though. These are mentioned briefly in table 1.

Statistic	Original	Renumbered
L1 Cache Line Reuse	5.42	5.67
L2 Cache Line Reuse	4548.95	9530.24
Memory bandwidth used (MB/s)	1.21	0.43

Table 1: Reduced memory system demands.

3.2.2 Compiler and Source Code Optimizations

Developers of scientific computing codes are quite familiar with the issues of speed and accuracy in numerical computing. In particular, of the many floating point operations possible, division is one of the most expensive in terms of cycles to completion. Within the scope of accuracy requirements, transformations such as strength reduction can give major performance gains. This technique involves replacing an expensive instruction with a series of more simple, quickly computed instructions. A good example is replacing a multiplication operation by a series of additions.

Prior to full optimization, the COMPOSE solver update routines contained several division operations. An analysis of the assembly code showed one case where a delay of nine unused cycles was being generated waiting for a division operation to complete. Further compounding this delay was the fact that it was positioned inside of a loop structure which in turn was nullifying any improvements from the software pipeliner. By using multiply by reciprocal, loop invariant code motion, and strength reduction, the original loop generating poor results

```
#<swps>   2 flops        (  7% of peak)
                     (madds count as 2)
#<swps>   0 madds        (  0% of peak)
#<swps>   3 mem refs     ( 21% of peak)
#<swps>   3 integer ops  ( 10% of peak)
#<swps>   8 instructions ( 14% of peak)
```

was replaced with code generating much better performance

```
#<swps>   8 flops        ( 30% of peak)
                     (madds count as 2)
#<swps>   4 madds        ( 30% of peak)
#<swps>  13 mem refs     (100% of peak)
#<swps>   6 integer ops  ( 23% of peak)
#<swps>  23 instructions ( 44% of peak)
```

This quick change reduced execution time by about 10%. Full-scale optimization and profiling will often reveal these potential areas of savings.

One final optimization employed is something we call reverse procedure integration. It acts in a reverse fashion from inlining. Basically, contiguous code segments that are internal to a subroutine are moved to their own subroutine. One of the most complicated tasks in compiler science is instruction scheduling. This technique serves as a way for the software developer to impart some importance on the code segment to the compiler and simplify the instruction scheduling pass.

The same analysis that revealed the issue of several no-op instructions in the division case mentioned earlier also revealed very complicated instruction scheduling in a critical region of the code. It was discovered that the compiler was generating code for a matrix-vector multiply that was executing at peak (in terms of instruction scheduled peak FLOP rates and memory references), but was very complex and was interweaving instructions from surrounding code. The out-of-order execution on the SGI Origin platform could be one cause of these instruction schemes. The outer loop, which was itself part of a larger loop structure, was being pipelined separately from the innermost loop. This seemed to stem from some sort of prefetching of values in the outer loop of the matrix-vector multiply. This interplay between optimizations (pipelining and prefetching) seemed to generate a fair amount of pressure on the instruction scheduler, and in fact appeared to be detrimental. Through reverse procedure integration, we have noted reductions of 15% in execution time on the SGI systems. The Cray system showed no changes, and the IBM system was not fully tested due to time constraints. We hope to visit this again for a more detailed analysis.

4 HPF

The High Performance Fortran language has been in existence for roughly the same amount of time as the MPI libraries. However, HPF has failed to capture the wide following of MPI for several reasons. One is the performance of the code generated by HPF. Since HPF is a language, code development is at a higher level, with low-level details left to the compiler. Also, some researchers no doubt incorrectly assume that data parallelism is not relevant on today's multiple instruction, multiple data computers. HPF compilers can generate code that performs well on shared memory, distributed shared memory, and cluster computers.

We have noted, however, that getting good performance with HPF requires time-consuming attention to low-level detail. The HPF compiler used for this software development actually acts as a front end translator that takes HPF code and generates Fortran code (either FORTRAN 77 or Fortran 90/95) and in turn passes this through the computer's native compiler. Communication can be completed with any of several techniques, such as proprietary libraries, or even through message passing with MPI. To truly understand the actions of the HPF compiler, it is often necessary to carefully analyze this underlying code.

4.1 Optimizations

We have implemented several strategies to boost overall HPF performance of this code. Most performance gains have resulted from reusing communication schedules in the matrix-vector operations, and from reorganizing the data to be more processor-partition friendly. A complete discussion is available in a previous report [10] and will not be revisited here.

A new development did surface in HPF while collecting data for this report. The Portland Group released version 3.2 of their HPF compiler. Just as in version 3.0, this new compiler allows for asymmetric block distribution of data. However, the new version is not constrained to work only with conformable arrays. The new version should be beneficial to unstructured grids where the pertinent arrays will never be conformable. The new version of the compiler will allow for shared memory compilation of asymmetric blocks rather than using replication.

Accordingly, we modified our preprocessing pass that does domain reconditioning to report the number of nodes and elements that are completely local to a processor. Each mesh is processed for a certain number of processors, and a vector is generated that tells the number of nodes and elements for each processor. The data is then distributed in varying sizes, thus stopping all but the essential communication between processors. This approach should allow for the most accurate comparison between HPF and MPI since each method will have the minimal amount of communication required.

Unfortunately, due to the short time span between the new release of the HPF compiler and the deadlines for this paper and several problems experienced with interactions between the HPF compiler and the native Fortran compiler on the Cray T3E system, we are currently unable to use this new approach in these studies. We will revisit this in the future.

5 Performance

5.1 HPC Platforms

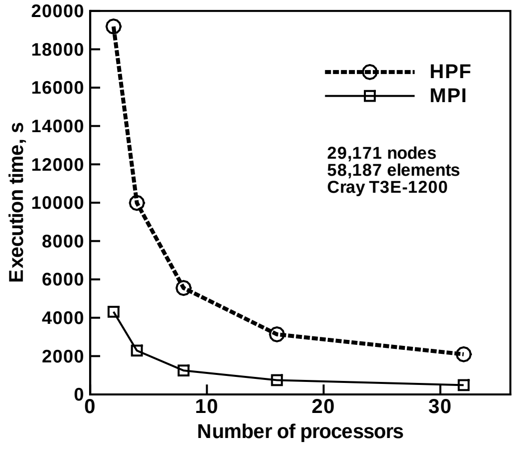
Several high performance computing (HPC) systems were used in this study. The Cray T3E-1200, a massively parallel (MPP) distributed memory platform, was used. This system, located at the U. S. Army High Performance Computing Research Center (Minneapolis, MN), has over 1024 processors with 500 MB of memory per processor. Two computers at the U. S. Army Research Laboratory's Major Shared Resource Center (MSRC) located at Aberdeen Proving Ground, MD were also utilized. The first is the SGI Origin 3800 system. At the time of the study, this machine consisted of 128 400-MHz R12000 processors. Recently, this computer was joined with another similar system to form a 256-processor parallel computer. The timing studies following only scale to the 128-processor configuration. The second computer is the IBM Nighthawk 2 SMP system. This configuration is comprised of 32 Power3-II SMP "high nodes," each with 16 processors clocked at 375 MHz (512 processors total).

5.2 Timing Results

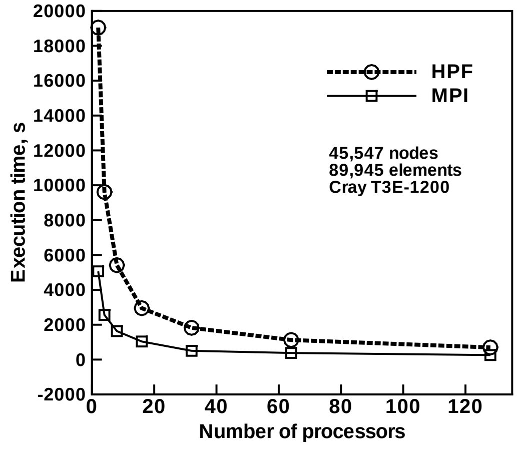
5.2.1 HPF and MPI Comparison

Due to some of the constraints mentioned earlier, the discussion of HPF versus MPI performance is limited to the Cray T3E system. We are currently investigating the use of the Portland Group HPF compiler on the SGI 3800 system. Those results will be presented as they become available.

The wall clock run times for two problem sets are shown in Figure 4. These times are exclusive of file input/output. It is worth noting that simulation compute time is related



(a) Mesh 1



(b) Mesh 2

Figure 4: Total execution time comparison of HPF and MPI scalable software.

both to mesh size and the parameter set being used (which were different in this case). The PGHPF software was compiled with the `-fast` option, which in turn causes the native Fortran compiler to be called with the `-Ounroll -Opipeline2 -Oscalar3` flags. The MPI Fortran 90 code was compiled with the `-O3,pipeline3` options enabled.

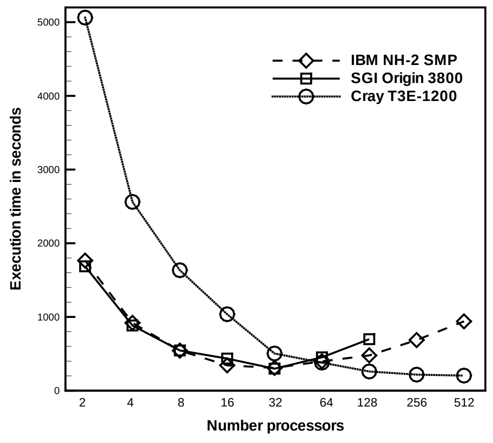
It can easily be seen that the MPI code outperformed its HPF counterpart in every trial. In Figure 4(a), the 2-processor HPF code required roughly 4.5 times longer to complete than the MPI version. At 32 processors, the MPI version outperformed the HPF version by about the same factor. In Figure 4(b), the 2-processor MPI code completed about 3.8 times faster than the similar HPF code. Although the gap started to shrink, MPI was still running about 2.7 times faster than the HPF code at 128 processors.

The meshes used in the HPF study were not renumbered for cache efficiency due to a lack of time. The mesh was renumbered to promote processor locality, but the meshes inside the processors were not renumbered for cache behavior. It is not anticipated that this will make a large difference in the execution time. While the RCM pass resulted in about a 10% savings on the MPI code, this savings is not enough to equalize performance between HPF and MPI.

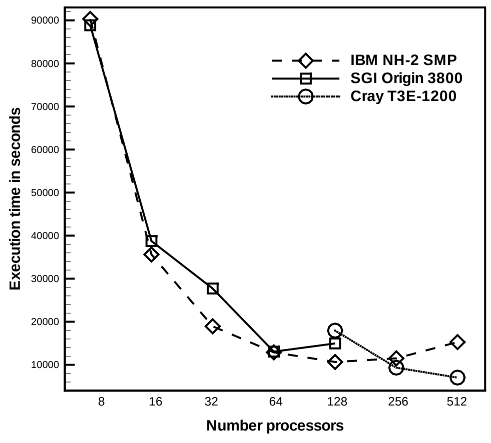
5.2.2 MPI Cross Platform Comparisons

The times reported in this section are based on jobs submitted in standard production run queues. The only exception is the IBM system that was in pioneer user mode during these trials. However, the machine was still operating at or near capacity.

Figure 5 shows the execution time of COMPOSE for a representative unstructured mesh. Figure 5(a) is for a mesh comprising 45,547 nodes and 89,945 elements, and Figure 5(b) is



(a) Mesh 1



(b) Mesh 4

Figure 5: Total execution time comparison of MPI scalable software.

for a mesh built from 405,327 nodes and 809,505 elements. The T3E plots in Figure 5(b) are not shown for less than 128 processors due to the execution time limit constraints set by the system's administrators. Wall clock time limits, regardless of the number of processors used, are set to 8 hours. We chose not to use restart files for these cases.

In Figure 5(b), it can be seen that SGI performance remained good up to 64 processors, and IBM performance remained good up to 128 processors. At this point, the performance curves start to bend upwards, indicating a switchover from being computation bound to communication bound. This was not unexpected since we were well aware of the demanding nature of communications required in the solver. Still, a 1-processor run for this case, which would require over two weeks to complete, can be completed on 64 processors in slightly over 3.5 hours.

The performance of the T3E system provided interesting commentary. In all cases, this platform required nearly double the number of CPUs as the SGI or IBM to achieve the same execution times. Only at large numbers of CPUs did the T3E outperform the others. This may also provide some insightful cost versus performance metrics for architectures required to perform many unstructured grid computations. Furthermore, the computational curve never bends, so this code never became communication bound on the Cray. This is probably the result of combining a highly tuned communication system and poor CPU performance for unstructured codes. The overly small cache in this system no doubt accounts for the latter.

This behavior is also seen in the NAS parallel benchmarks. Table 2 lists the processor MFLOP performance of the block tridiagonal (BT) solver and conjugate gradient (CG) benchmarks. This data is gathered from several sources [11–13]. BT represents a structured grid problem, while CG represents a kernel that is typical of unstructured grid computations

System	Clock (MHz)	MFLOPS		Ratio $\frac{bt}{cg}$
		BT	CG	
SGI Origin 2000	195	55	28	1.96
IBM Power3 High	222	116	47	2.47
Cray T3E-1200	600	164	26	6.30

Table 2: Relative performance of structured versus unstructured codes.

with irregular data access patterns. Notice that the T3E has a much higher BT to CG ratio of MFLOP performance. This illustrates that unstructured grid computations are not nearly as efficient on that machine as they are on the IBM and SGI systems.

6 Concluding Remarks

This paper presented performance results from an application that has benefited significantly from parallel computing. Computing times requiring weeks can be shortened to hours, thus enabling comprehensive and timely modeling and simulation results. Both HPF and MPI represent viable options in unstructured grid computations. Both approaches have shown reduced wall clock times as more processors are added to the problem. HPF lags behind MPI in performance for this unstructured code, and with the large attention to detail required to get this performance, does not really provide any easier formulation than the low-level message passing approach. However, the features being added to the HPF language, and the maturity of the compilers, shows continued improvement.

The in-processor cache demands and repeated global reductions in this code provide a good mixture to test the computational and communication capabilities of various systems. Good performance requires the best of all worlds–efficient compilers, fast processor interconnection, large caches, clock speeds, etc. Currently, for this unstructured grid computation we see the SMP-based IBM and SGI architectures outperforming the distributed memory Cray T3E at all but the largest CPU-set levels. Undoubtedly, this is mostly due to the tradeoffs made by the computer engineers in providing fast interconnections with small caches.

We have shown how careful attention to detail, profiling, and performance analysis can help generate good efficiency. The code itself highlights the importance and need for good solution approaches such as in-node reductions prior to global reductions to reduce communication requirements. This application should serve as a good test case for the next generation of computers with enhanced cache and interconnection networks.

7 Acknowledgments

This research was made possible by a grant of computer time and resources by the Department of Defense High Performance Computing Modernization Program and CHSSI project.

References

[1] R. V. Mohan, N. D. Ngo, and K. K. Tamma. On a Pure Finite-Element-Based Methodology for Resin Transfer Mold Filling Simulations. *Polymer Engineering and Science*, 39(1), January 1999.

[2] Ram Mohan, Dale Shires, and Andrew Mark. The Application of Process Simulations Tools to Reduce Risks in Liquid Molding of Composites. American Helicopter Society Annual Forum, May 2001.

[3] Ram Mohan, Dale Shires, and Andrew Mark. Large Scale Process Modeling Simulations in Liquid Composite Molding. International Conference on Computational Science, May 2001.

[4] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.* University of Minnesota and the Army HPC Research Center, 1997.

[5] C. Walshaw. *JOSTLE: Mesh Partitioning Software.* University of Greenwich, London, England. Hypertext at http://www.gre.ac.uk/ jjg01/.

[6] K. H. Law. A Parallel Finite Element Solution Method. *Computers & Structures*, 23(6):845 – 858, 1985.

[7] R. Kanapady. Parallel Implementation of Large Scale Finite Element Computations on a Multiprocessor Machine: Applications to Process Modeling and Manufacturing of Composites. Master's thesis, University of Minnesota, 1998.

[8] G. Kumfert and A. Pothen. Two Improved Algorithms for Envelope and Wavefront Reduction. *BIT*, 37(3):559 – 590, 1997.

[9] E. Cuthill and J. McKee. Reducing the Bandwidth of Sparse Symmetric Matrices. In *24th National Conference*, pages 157–172. Association for Computing Machinery, 1969.

[10] Dale Shires, Ram Mohan, and Andrew Mark. Improving Data Locality and Expanding the Use of HPF in Parallel FEM Implementations. International Conference on Parallel and Distributed Processing Techniques and Applications, 2000.

[11] NAS Parallel Benchmarks 2 website, 1997. Numerical Aerospace Simulation Group, NASA.

[12] E. Anderson. Performance Analysis of the Cray T3E-1200E. Technical report, National Environmental Supercomputing Center, 2000.

[13] M. Barrios, S. Andersson, and G. Bhanot. Scientific Applications in RS/6000 SP Environments. Technical report, IBM, 1999. Redbook SG24-5611-00.